Streaming SIMD Extensions – Deformable Surfaces

Version 1.1

01/99

Order Number: 243631-004

Table of Contents

Revision History

Revision	Revision History	Date
1.1	FCS revision.	01-99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1 P. Volino, N. M. Thalmann, 1997, *"Developing Simulation Techniques for an Interactive Clothing System"*, MIRALab, University of Geneva, http://miralabwww.unige.ch.

2 *"Geri's Game"*, 1998, Pixar Animation Studios, http://www.pixar.com.

1 Introduction

Rigid objects limit realism in games. Deformable surfaces allow the user to interact more realistically with an environment and provide visual feedback to the user about the organic nature of objects, such as water, cloth, and mountains. Typical examples of visual feedback include explosions causing craters in the ground, stones causing ripples in water, and wind causing a cloth cape to flutter. By deforming or modifying the underlying geometry of an object, and not merely mapping a moving texture onto it, the user receives a much richer sense of motion and set of visual cues from the environment.

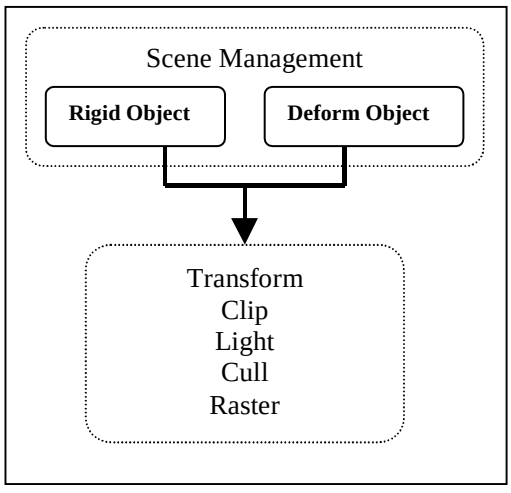
The implementation of deformable surfaces discussed in this app note is intended for real-time applications such as 3D games. The algorithm presented is used to create real-time water and cloth, as well as giving polygonal models a "rubber-like", deformable property. It is a subset of a more complex and non-realtime set of algorithms for simulating realistic cloth [1]. An example of the commercial use of deformable surfaces is the proprietary cloth simulator developed by Pixar for a short animation titled "Geri's Game" [2].

2 Algorithm – Deformable Surfaces

A typical graphics pipeline is shown in Figure 1. The Scene Manager layer sends visible objects through the pipeline. The reference copy of a rigid object's vertex and normal data are static in model space. Perspective transformations from model to screen space are applied to the vertex and normal data, leaving the original shape of the object intact. For rigid objects, vertices have the same geometric relationship to each other before and after the transformation.

For objects with deformable surfaces, the vertex and normal are modified in model space before applying the transformation. The direction and distance between neighboring vertices is modified, and normal vectors are recalculated.

Figure 1. Typical graphics pipeline with rigid and deformable surfaces.



2.1 Background

A copy of the original topology and shape of the deformable surface is kept as a reference for calculating internal elastic forces within the surface. Two possible cases are:

1) The reference data is not updated as the surface deforms. The resulting object returns to its original shape when no external forces are applied. Ripples on a water surface are an example.

2) The reference data is periodically modified to follow deformations. The resulting surface is ductile. Craters in a terrain surface caused by an explosion are an example.

External forces from objects that push or pull on deformable surface vertices are known as effectors. Effectors modify the accelerations, and thus positions, of the vertices at each iteration of the deformation simulation. Varying the forces exerted by the effectors and defining elasticity between deformable surface vertices produces various fluid-like properties. Surfaces representing water, deformable terrain, cloth, or any mesh based character or object with "rubber-like" properties can be deformed using this algorithm.

2.2 Function of the Algorithm

For each deformable surface vertex, all external and internal forces are summed. Figure 2 shows an effector exerting an external force on a vertex a time t=0, t=DeltaT, and t=2DeltaT.

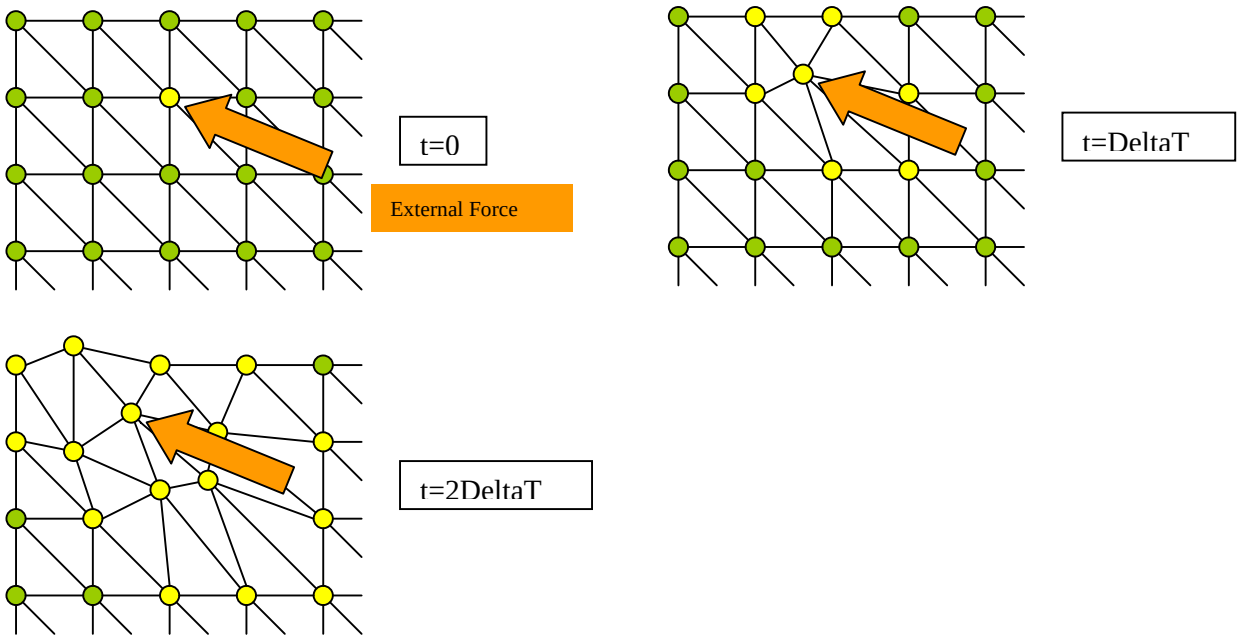


Figure 2. An effector exerting external force on a deformable surface at times T, DeltaT, and 2DeltaT

The internal forces known as DeltaElastic is the difference between the original reference position and the current vertex position, generated according to a simple linear elastic model where:

InternalForce = ElasticConstant * DeltaElastic

This relationship is illustrated in Figure 3.

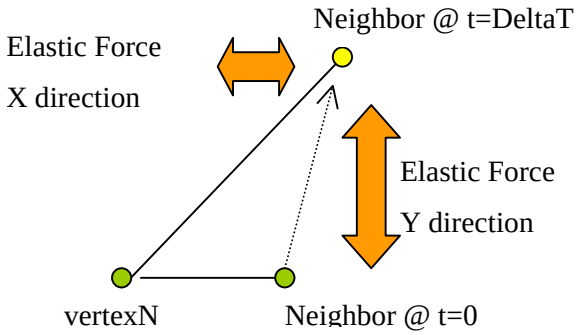


Figure 3. Linear Elastic Model

As the distance between neighboring vertices varies, according to the mesh topology, positive and negative forces are created between these vertices. Figure 3 shows a larger force in the Y direction corresponding to a larger vertex displacement in the Y direction. For each vertex in the model, all forces from neighboring vertices are summed.

Using Newton's Second Law of motion,

Force = Mass * Acceleration

the acceleration of each vertex is calculated. The simulation iterates by a delta of time t = DeltaT. Applying DeltaT to acceleration yields, the change in velocity, Delta V:

DeltaV = Acceleration * DeltaT.

The change in position, DeltaXYZ, is given by:

DeltaXYZ = DeltaV * DeltaT.

Adding DeltaXYZ to the current vertex position yields the position for the next iteration.

For this paper, a rectangular planar surface with a topology defined by 6 nearest neighbor vertices was chosen. Each vertex is connected to its nearest neighbors in the North, East, Southeast, South, West, and Northwest directions. A SIMD version of the algorithm is easily defined for each vertex in this planar deformable surface by calculating the internal forces between its neighbors to the East, Southeast, and South. The negative of each force is added to the corresponding neighbor's total force. As an example, the internal force between a vertex and its East neighbor is the negative of the force between the East neighbor and the vertex.

3 Performance

Streaming SIMD Extensions enable increased performance over scalar floating-point code, through two classes of operations: prefetch and parallel arithmetic operations.

For a regular mesh with similar topology between neighboring vertices, the components of each vertex, such as position, acceleration, etc., can be arranged in a Structure of Arrays (SoA) format. The SoA format arranges similar data in contiguous blocks of memory. For example, an SoA format could arrange one array for all vertex X position values, another array for all Y position values, and another array for all Z position values. With the SoA format, the computations for four vertices are performed in parallel.

The topology for the planar deformable surface described in this application note is defined such that all neighboring connections are evaluated by operating on vertices to the East, SouthEast, and South. The processing starts at the upper left corner of the rectangular mesh and proceeds in a column then row order to the lower right corner. For each vertex, computations for force, acceleration, velocity, etc. are performed on only the vertex neighbors to the East, SouthEast, and South directions. Memory access patterns are known a-priori and suitable to prefetch.

4 Conclusion

The Ssemai New Instructions enable higher performance for deformable surface applications. The higher performance is a result of operating on the vertex components of a planar deformable surface, arranged in a SoA format, in a SIMD manner. For the regular topology structure used in the test application, the prefetch instruction overlapped memory accesses with computations, thus hiding part of the memory latency and access time.

5 Deformable Surfaces Code Examples

C and Ssemai New Instructions versions of the algorithm for a planar topology with 6 nearest neighbors are shown below. The two main functions are:

1) Calculate the collision of a sphere with surface vertices (external forces).

2) Update vertex position using external and internal elastic forces.

The collision function is called for each effector. Effectors are rigid objects, such as a sphere for this implementation, which have a pushing or pulling capability. After calculating all external forces, the update function generates the new deformable surface vertex positions.

`\planar_deform.c`	C and Ssemai New Instructions code implementations.
`\deform1.bat`	Runs the application with configuration for a water surface.

5.1 C Code for Deformable Surfaces

The following C code performs the collision detection of a sphere with a planar deformable surface.

```
void CollisionPlanarDeformObject(PlanarDeformObject* Deform, RigidObject* Effector)
{
int     i;
int     j;
int     index;
float   x;
float   y;
float   z;
float   distance;
float   length;
float   NewForceDirX;
float   NewForceDirY;
float   NewForceDirZ;
float   InfluenceRadiusSqrd;
Vector4 CenterOffset;

InfluenceRadiusSqrd = Effector->InfluenceRadius * Effector->InfluenceRadius;
CenterOffset.x  = Effector->Center.x - Deform->Center.x;
CenterOffset.y  = Effector->Center.y - Deform->Center.y;
CenterOffset.z  = Effector->Center.z - Deform->Center.z;
```

```
for(i=0; i<Deform->NumRows; i++)
{
  for(j=0; j<Deform->NumCols; j++)
  {
   index = i*Deform->NumCols + j;
   x = CenterOffset.x -  Deform->VertexX[index];
   y = CenterOffset.y -  Deform->VertexY[index];
   z = CenterOffset.z -  Deform->VertexZ[index];
   distance = x*x + y*y + z*z;
   if(distance < InfluenceRadiusSqrd)
   {
     length = sqrt(distance);
     NewForceDirX = (x*Effector->Strength) / (length+0.01);
     NewForceDirY = (y*Effector->Strength) / (length+0.01);
     NewForceDirZ = (z*Effector->Strength) / (length+0.01);

     Deform->ExternalForceDirX[index]+= NewForceDirX;
     Deform->ExternalForceDirY[index]+= NewForceDirY;
     Deform->ExternalForceDirZ[index]+= NewForceDirZ;
   }
 }
}
}
```

The following C code updates the acceleration, velocity, and position components for each planar deformable surface vertex, based on collision and internal elastic forces.

```
void UpdatePlanarDeformObject(PlanarDeformObject* Deform)
{
int   i;
int   j;
int   k;
int   stride;
int   IndexCenter;
int   Index[8];
int   index;
float x;
float y;
float z;
float distance;
float U_x;
float U_y;
float U_z;
float dx;
float dy;
```

```
float dz;
float dfx;
float dfy;
float dfz;
float AccelerationX;
float AccelerationY;
float AccelerationZ;
float atten;
k = Deform->NumRows*Deform->NumCols;
for(i=0; i<k; i++)
{
  Deform->TotalForceDirX[i] = 0.0;
  Deform->TotalForceDirY[i] = 0.0;
  Deform->TotalForceDirZ[i] = 0.0;
}
stride = Deform->NumCols;
for(i=0; i<Deform->NumRows-1; i++)
{
  for(j=0; j<Deform->NumCols-1; j++)
  {
    index = i*stride + j;
    dx  = Deform->VertexX[index+1] - Deform->VertexX[index];
    dy  = Deform->VertexY[index+1] - Deform->VertexY[index];
    dz  = Deform->VertexZ[index+1] - Deform->VertexZ[index];

    dx -= Deform->RestLengthE.x;
    dy -= Deform->RestLengthE.y;
    dz -= Deform->RestLengthE.z;
    dfx  = Deform->Elasticity * dx;
    dfy  = Deform->Elasticity * dy;
    dfz  = Deform->Elasticity * dz;

    Deform->TotalForceDirX[index] += dfx;
    Deform->TotalForceDirY[index] += dfy;
    Deform->TotalForceDirZ[index] += dfz;

    Deform->TotalForceDirX[index+1]-= dfx;
    Deform->TotalForceDirY[index+1]-= dfy;
    Deform->TotalForceDirZ[index+1]-= dfz;

    dx = Deform->VertexX[index+stride]- Deform->VertexX[index];
    dy = Deform->VertexY[index+stride]- Deform->VertexY[index];
    dz = Deform->VertexZ[index+stride]- Deform->VertexZ[index];

    dx -= Deform->RestLengthS.x;
```

```
    dy -= Deform->RestLengthS.y;
    dz -= Deform->RestLengthS.z;

    dfx  = Deform->Elasticity * dx;
    dfy  = Deform->Elasticity * dy;
    dfz  = Deform->Elasticity * dz;

    Deform->TotalForceDirX[index] += dfx;
    Deform->TotalForceDirY[index] += dfy;
    Deform->TotalForceDirZ[index] += dfz;

    Deform->TotalForceDirX[index+stride]-= dfx;
    Deform->TotalForceDirY[index+stride]-= dfy;
    Deform->TotalForceDirZ[index+stride]-= dfz;

    dx = Deform->VertexX[index+stride+1]- Deform->VertexX[index];
    dy = Deform->VertexY[index+stride+1]- Deform->VertexY[index];
    dz = Deform->VertexZ[index+stride+1]- Deform->VertexZ[index];

    dx -= Deform->RestLengthSE.x;
    dy -= Deform->RestLengthSE.y;
    dz -= Deform->RestLengthSE.z;

    dfx  = Deform->Elasticity * dx;
    dfy  = Deform->Elasticity * dy;
    dfz  = Deform->Elasticity * dz;

    Deform->TotalForceDirX[index] += dfx;
    Deform->TotalForceDirY[index] += dfy;
    Deform->TotalForceDirZ[index] += dfz;

    Deform->TotalForceDirX[index+stride+1] -= dfx;
    Deform->TotalForceDirY[index+stride+1] -= dfy;
    Deform->TotalForceDirZ[index+stride+1] -= dfz;
  }
}
k = Deform->NumRows*Deform->NumCols;
for(i=0; i<k; i++)
{
  Deform->TotalForceDirX[i] += Deform->ExternalForceDirX[i];
  Deform->TotalForceDirY[i] += Deform->ExternalForceDirY[i];
  Deform->TotalForceDirZ[i] += Deform->ExternalForceDirZ[i];

  Deform->TotalForceDirX[i] += Deform->ConstantForceDirX[i];
  Deform->TotalForceDirY[i] += Deform->ConstantForceDirY[i];
```

```
  Deform->TotalForceDirZ[i] += Deform->ConstantForceDirZ[i];

  if(!Deform->Fixed[i])
  {
    AccelerationX = Deform->TotalForceDirX[i] / Deform->Mass;
    AccelerationY = Deform->TotalForceDirY[i] / Deform->Mass;
    AccelerationZ = Deform->TotalForceDirZ[i] / Deform->Mass;

    Deform->VelocityX[i] += AccelerationX * DeltaT;
    Deform->VelocityY[i] += AccelerationY * DeltaT;
    Deform->VelocityZ[i] += AccelerationZ * DeltaT;

    Deform->VertexX[i] += Deform->VelocityX[i] * DeltaT;
    Deform->VertexY[i] += Deform->VelocityY[i] * DeltaT;
    Deform->VertexZ[i] += Deform->VelocityZ[i] * DeltaT;

    atten = 1.0 - Deform->VelocityY[i];
    if(atten < 0.7) atten = 0.7;
    Deform->WorkingColor[i].r  = Deform->Color[i].r * (atten);
    Deform->WorkingColor[i].g  = Deform->Color[i].g * (atten);
    Deform->WorkingColor[i].b  = Deform->Color[i].b * (atten);
    Deform->WorkingColor[i].a  = Deform->Color[i].a;
    Deform->VelocityX[i] *= Deform->DampingFactor;
    Deform->VelocityY[i] *= Deform->DampingFactor;
    Deform->VelocityZ[i] *= Deform->DampingFactor;
    Deform->WorkingNormal[i].x = Deform->VelocityX[i];
    Deform->WorkingNormal[i].y = Deform->VelocityY[i];
    Deform->WorkingNormal[i].z = Deform->VelocityZ[i];
    Deform->ExternalForceDirX[i] = 0.0;
    Deform->ExternalForceDirY[i] = 0.0;
    Deform->ExternalForceDirZ[i] = 0.0;
  }
  Deform->WorkingVertex[i].x  = Deform->VertexX[i] + Deform->Center.x;
  Deform->WorkingVertex[i].y  = Deform->VertexY[i] + Deform->Center.y;
  Deform->WorkingVertex[i].z  = Deform->VertexZ[i] + Deform->Center.z;
  Deform->WorkingVertex[i].w  = 1.0;
}
}
```

5.2 Streaming SIMD Extensions for Deformable Surfaces (Intrinsics)

The following code using Streaming SIMD Extensions performs the collision detection of a sphere with a planar deformable surface. This section shows an example of deformable surfaces, using the Intrinsics form of the Streaming SIMD Extensions.

```
void CollisionPlanarDeformObject_SSE(PlanarDeformObject* Deform, RigidObject* Effector)
{
int     i;
int     j;
int     k;
int     m;
int     len;
int     mask=0x7FFFFFFF;
int     lenX;
int     lenY;
int     lenZ;
__m128  distanceX;
__m128  distanceY;
__m128  distanceZ;
__m128  x;
__m128  y;
__m128  z;
__m128  distance;
__m128  length;
__m128  temp;
__m128  InfluenceRadiusSqrd;
__m128  NewForceDirX_sse;
__m128  NewForceDirY_sse;
__m128  NewForceDirZ_sse;
__m128  SignMask = _mm_set_ps1(*((float*)&mask));
__m128  CenterOffsetX  = _mm_set_ps1(Effector->Center.x - Deform->Center.x);
__m128  CenterOffsetY  = _mm_set_ps1(Effector->Center.y - Deform->Center.y);
__m128  CenterOffsetZ  = _mm_set_ps1(Effector->Center.z - Deform->Center.z);
__m128  Strength_sse   = _mm_set_ps1(Effector->Strength);
__m128  point_zero_one = _mm_set_ps1(0.01);
__m128  TempObjectExternalForceDirX = _mm_set_ps1(0.0);
__m128  TempObjectExternalForceDirY = _mm_set_ps1(0.0);
__m128  TempObjectExternalForceDirZ = _mm_set_ps1(0.0);
__m128 *VertexX_sse;
__m128 *VertexY_sse;
__m128 *VertexZ_sse;
__m128 *ExForceDirX_sse;
__m128 *ExForceDirY_sse;
__m128 *ExForceDirZ_sse;

InfluenceRadiusSqrd = _mm_load_ps1(&Effector->InfluenceRadius);
InfluenceRadiusSqrd = _mm_mul_ps(InfluenceRadiusSqrd, InfluenceRadiusSqrd);

VertexX_sse = (__m128 *)Deform->VertexX;
VertexY_sse = (__m128 *)Deform->VertexY;
```

```
VertexZ_sse = (__m128 *)Deform->VertexZ;

ExForceDirX_sse = (__m128 *)Deform->ExternalForceDirX;
ExForceDirY_sse = (__m128 *)Deform->ExternalForceDirY;
ExForceDirZ_sse = (__m128 *)Deform->ExternalForceDirZ;

k = Deform->NumRows*Deform->NumCols/4;
for(i=0; i<k; i++)
{
  x = _mm_sub_ps(CenterOffsetX, VertexX_sse[i]);
  y = _mm_sub_ps(CenterOffsetY, VertexY_sse[i]);
  z = _mm_sub_ps(CenterOffsetZ, VertexZ_sse[i]);

  distanceX = _mm_mul_ps(x,x);
  distanceY = _mm_mul_ps(y,y);
  lenX = _mm_movemask_ps(_mm_cmplt_ps(distanceX, InfluenceRadiusSqrd));
  if (lenX == 0) continue;
  lenY = _mm_movemask_ps(_mm_cmplt_ps(distanceY, InfluenceRadiusSqrd));
  if (lenY == 0) continue;
  distance = _mm_add_ps(distanceX, _mm_add_ps(distanceY, _mm_mul_ps(z,z)));
  len = _mm_movemask_ps(_mm_cmplt_ps(distance, InfluenceRadiusSqrd));
  if (len != 0)
  {
    temp   = _mm_cmplt_ps(distance, InfluenceRadiusSqrd);
    length = _mm_sqrt_ps(distance);
    NewForceDirX_sse = _mm_and_ps(_mm_div_ps(_mm_mul_ps(x, Strength_sse),
                                             _mm_add_ps(length, point_zero_one)),
                                  temp);
    NewForceDirY_sse = _mm_and_ps(_mm_div_ps(_mm_mul_ps(y, Strength_sse),
                                             _mm_add_ps(length, point_zero_one)),
                                  temp);
    NewForceDirZ_sse = _mm_and_ps(_mm_div_ps(_mm_mul_ps(z, Strength_sse),
                                             _mm_add_ps(length, point_zero_one)),
                                  temp);
   ExForceDirX_sse[i] = _mm_add_ps(ExForceDirX_sse[i], NewForceDirX_sse);
   ExForceDirY_sse[i] = _mm_add_ps(ExForceDirY_sse[i], NewForceDirY_sse);
   ExForceDirZ_sse[i] = _mm_add_ps(ExForceDirZ_sse[i], NewForceDirZ_sse);


for(m = 0; m<4; m++)
{
  Effector->ExternalForceDir.x -= ((float *)(&TempObjectExternalForceDirX))[m];
  Effector->ExternalForceDir.y -= ((float *)(&TempObjectExternalForceDirY))[m];
  Effector->ExternalForceDir.z -= ((float *)(&TempObjectExternalForceDirZ))[m];
}
```

```
}
```

The following code using Streaming SIMD Extensions updates the acceleration, velocity, and position components for each planar deformable surface vertex, based on collision and internal elastic forces.

```
void UpdatePlanarDeformObject_sse(PlanarDeformObject* Deform)
{
__m128  FixedMask;
__m128  WriteMask;
__m128  OnesMask[4];
__m128  Mass_sse;
__m128  diffX;
__m128  diffY;
__m128  diffZ;
__m128  diff_forceX;
__m128  diff_forceY;
__m128  diff_forceZ;
__m128  Elasticity_sse;
__m128  DeltaT_sse = _mm_set_ps1(DeltaT);
__m128  DampingFactor_sse = _mm_set_ps1(Deform->DampingFactor);
__m128  CenterX  = _mm_set_ps1(Deform->Center.x);
__m128  CenterY  = _mm_set_ps1(Deform->Center.y);
__m128  CenterZ  = _mm_set_ps1(Deform->Center.z);
__m128  AllZeros = _mm_set_ps1(0.0);
__m128  LowerBound = _mm_set_ps(0.7, 0.7, 0.7, 0.7);
__m128  UpperBound = _mm_set_ps(1.0, 1.0, 1.0, 1.0);
__m128  OnePointZero = _mm_set_ps(1.0, 1.0, 1.0, 1.0);
__m128  temp;
__m128  tempX;
__m128  tempY;
__m128  tempZ;
__m128  RestLengthXE_sse;
__m128  RestLengthXS_sse;
__m128  RestLengthXSE_sse;
__m128  RestLengthYE_sse;
__m128  RestLengthYS_sse;
__m128  RestLengthYSE_sse;
__m128  RestLengthZE_sse;
__m128  RestLengthZS_sse;
__m128  RestLengthZSE_sse;
__m128  wv0;
__m128  wv1;
__m128  wv2;
__m128  wv3;
```

```
__m128  tempVX;
__m128  tempVY;
__m128  tempVZ;
__m128  AccelerationX_sse;
__m128  AccelerationY_sse;
__m128  AccelerationZ_sse;
__m128 *TotalForceDirX_sse;
__m128 *TotalForceDirY_sse;
__m128 *TotalForceDirZ_sse;
__m128 *VertexX_sse;
__m128 *VertexY_sse;
__m128 *VertexZ_sse;
__m128 *ExternalForceDirX_sse;
__m128 *ExternalForceDirY_sse;
__m128 *ExternalForceDirZ_sse;
__m128 *ConstantForceDirX_sse;
__m128 *ConstantForceDirY_sse;
__m128 *ConstantForceDirZ_sse;
__m128 *VelocityX_sse;
__m128 *VelocityY_sse;
__m128 *VelocityZ_sse;
__m128 *Color_sse;
__m128 *WorkColor_sse;
int     i;
int     j;
int     k;
int     stride;
int     Index;
int     Index4;
int     NeighborIndex;
int     FixedCompare;
float   atten;
float  *fVertexX_sse;
float  *fVertexY_sse;
float  *fVertexZ_sse;
float  *fTotalForceDirX_sse;
float  *fTotalForceDirY_sse;
float  *fTotalForceDirZ_sse;

TotalForceDirX_sse = (__m128 *)Deform->TotalForceDirX;
TotalForceDirY_sse = (__m128 *)Deform->TotalForceDirY;
TotalForceDirZ_sse = (__m128 *)Deform->TotalForceDirZ;

fTotalForceDirX_sse = (float *)TotalForceDirX_sse;
fTotalForceDirY_sse = (float *)TotalForceDirY_sse;
```

```
fTotalForceDirZ_sse = (float *)TotalForceDirZ_sse;

OnesMask[0] = _mm_set_ps1(0.0);
OnesMask[1] = _mm_set_ps1(0.0);
OnesMask[2] = _mm_set_ps1(0.0);
OnesMask[3] = _mm_set_ps1(0.0);
((int*)&OnesMask)[0]  = 0xFFFFFFFF;
((int*)&OnesMask)[1]  = 0xFFFFFFFF;
((int*)&OnesMask)[2]  = 0xFFFFFFFF;
((int*)&OnesMask)[3]  = 0xFFFFFFFF;
((int*)&OnesMask)[4]  = 0xFFFFFFFF;
((int*)&OnesMask)[8]  = 0xFFFFFFFF;
((int*)&OnesMask)[9]  = 0xFFFFFFFF;
((int*)&OnesMask)[12] = 0xFFFFFFFF;
((int*)&OnesMask)[13] = 0xFFFFFFFF;
((int*)&OnesMask)[14] = 0xFFFFFFFF;

k = Deform->NumRows*Deform->NumCols/4;
for(i=0; i<k; i++)
{
  TotalForceDirX_sse[i] = _mm_setzero_ps();
  TotalForceDirY_sse[i] = _mm_setzero_ps();
  TotalForceDirZ_sse[i] = _mm_setzero_ps();
}
Elasticity_sse   = _mm_set_ps1(Deform->Elasticity);

RestLengthXE_sse = _mm_set_ps1(Deform->RestLengthE.x);
RestLengthYE_sse = _mm_set_ps1(Deform->RestLengthE.y);
RestLengthZE_sse = _mm_set_ps1(Deform->RestLengthE.z);

RestLengthXS_sse = _mm_set_ps1(Deform->RestLengthS.x);
RestLengthYS_sse = _mm_set_ps1(Deform->RestLengthS.y);
RestLengthZS_sse = _mm_set_ps1(Deform->RestLengthS.z);

RestLengthXSE_sse = _mm_set_ps1(Deform->RestLengthSE.x);
RestLengthYSE_sse = _mm_set_ps1(Deform->RestLengthSE.y);
RestLengthZSE_sse = _mm_set_ps1(Deform->RestLengthSE.z);

VertexX_sse  = (__m128 *)Deform->VertexX;
VertexY_sse  = (__m128 *)Deform->VertexY;
VertexZ_sse  = (__m128 *)Deform->VertexZ;
fVertexX_sse = (float *)VertexX_sse;
fVertexY_sse = (float *)VertexY_sse;
fVertexZ_sse = (float *)VertexZ_sse;
```

```
Mass_sse     = _mm_set_ps1(Deform->Mass);

ExternalForceDirX_sse = (__m128 *)Deform->ExternalForceDirX;
ExternalForceDirY_sse = (__m128 *)Deform->ExternalForceDirY;
ExternalForceDirZ_sse = (__m128 *)Deform->ExternalForceDirZ;

ConstantForceDirX_sse = (__m128 *)Deform->ConstantForceDirX;
ConstantForceDirY_sse = (__m128 *)Deform->ConstantForceDirY;
ConstantForceDirZ_sse = (__m128 *)Deform->ConstantForceDirZ;

VelocityX_sse = (__m128 *)Deform->VelocityX;
VelocityY_sse = (__m128 *)Deform->VelocityY;
VelocityZ_sse = (__m128 *)Deform->VelocityZ;

WorkColor_sse = (__m128 *)Deform->WorkingColor;
Color_sse     = (__m128 *)Deform->Color;

stride = Deform->NumCols;
for(i=0; i<Deform->NumRows-1; i++)
{
  k = Deform->NumCols - 1;
  for(j=0, Index = i*stride, Index4 = (i*stride)>>2; j<k;  j+=4, Index += 4, Index4++)
  {
    if((k - j) > 3)
      WriteMask = OnesMask[0];
    else
      WriteMask = OnesMask[k-j];

    // Process vertices to the East
    NeighborIndex = Index + 1;
    tempX = _mm_loadu_ps((float*) &((float*)VertexX_sse)[NeighborIndex]);
    tempY = _mm_loadu_ps((float*) &((float*)VertexY_sse)[NeighborIndex]);
    tempZ = _mm_loadu_ps((float*) &((float*)VertexZ_sse)[NeighborIndex]);

    diffX = _mm_sub_ps(tempX, VertexX_sse[Index4]);
    diffY = _mm_sub_ps(tempY, VertexY_sse[Index4]);
    diffZ = _mm_sub_ps(tempZ, VertexZ_sse[Index4]);

    diffX = _mm_sub_ps(diffX, RestLengthXE_sse);
    diffY = _mm_sub_ps(diffY, RestLengthYE_sse);
    diffZ = _mm_sub_ps(diffZ, RestLengthZE_sse);

    diff_forceX = _mm_and_ps(WriteMask, _mm_mul_ps(Elasticity_sse, diffX));
    diff_forceY = _mm_and_ps(WriteMask, _mm_mul_ps(Elasticity_sse, diffY));
    diff_forceZ = _mm_and_ps(WriteMask, _mm_mul_ps(Elasticity_sse, diffZ));
```

```
TotalForceDirX_sse[Index4] = _mm_add_ps(TotalForceDirX_sse[Index4], diff_forceX);
TotalForceDirY_sse[Index4] = _mm_add_ps(TotalForceDirY_sse[Index4], diff_forceY);
TotalForceDirZ_sse[Index4] = _mm_add_ps(TotalForceDirZ_sse[Index4], diff_forceZ);

tempX = _mm_loadu_ps((float*) &TotalForceDirX_sse[NeighborIndex]);
tempY = _mm_loadu_ps((float*) &(TotalForceDirY_sse)[NeighborIndex]);
tempZ = _mm_loadu_ps((float*) &(TotalForceDirZ_sse)[NeighborIndex]);

tempX = _mm_sub_ps(tempX, diff_forceX);
tempY = _mm_sub_ps(tempY, diff_forceY);
tempZ = _mm_sub_ps(tempZ, diff_forceZ);

_mm_storeu_ps((float*)&(TotalForceDirX_sse)[NeighborIndex], tempX);
_mm_storeu_ps((float*)&(TotalForceDirY_sse)[NeighborIndex], tempY);
_mm_storeu_ps((float*)&(TotalForceDirZ_sse)[NeighborIndex],tempZ);

// Process vertices to the South
NeighborIndex = Index + stride;
tempX = _mm_loadu_ps((float*)&(float*)VertexX_sse)[NeighborIndex]);
tempY = _mm_loadu_ps((float*)&((float*)VertexY_sse)[NeighborIndex]);
tempZ = _mm_loadu_ps((float*)&((float*)VertexZ_sse)[NeighborIndex]);

diffX = _mm_sub_ps(tempX, VertexX_sse[Index4]);
diffY = _mm_sub_ps(tempY, VertexY_sse[Index4]);
diffZ = _mm_sub_ps(tempZ, VertexZ_sse[Index4]);
diffX = _mm_sub_ps(diffX, RestLengthXS_sse);
diffY = _mm_sub_ps(diffY, RestLengthYS_sse);
diffZ = _mm_sub_ps(diffZ, RestLengthZS_sse);

diff_forceX = _mm_and_ps(WriteMask, _mm_mul_ps(Elasticity_sse, diffX));
diff_forceY = _mm_and_ps(WriteMask, _mm_mul_ps(Elasticity_sse, diffY));
diff_forceZ = _mm_and_ps(WriteMask, _mm_mul_ps(Elasticity_sse, diffZ));

TotalForceDirX_sse[Index4] = _mm_add_ps(TotalForceDirX_sse[Index4], diff_forceX);
TotalForceDirY_sse[Index4] = _mm_add_ps(TotalForceDirY_sse[Index4], diff_forceY);
TotalForceDirZ_sse[Index4] = _mm_add_ps(TotalForceDirZ_sse[Index4], diff_forceZ);

tempX = _mm_loadu_ps((float*) &((float*)TotalForceDirX_sse)[NeighborIndex]);
tempY = _mm_loadu_ps((float*) &((float*)TotalForceDirY_sse)[NeighborIndex]);
tempZ = _mm_loadu_ps((float*) &((float*)TotalForceDirZ_sse)[NeighborIndex]);

tempX = _mm_sub_ps(tempX, diff_forceX);
tempY = _mm_sub_ps(tempY, diff_forceY);
tempZ = _mm_sub_ps(tempZ, diff_forceZ);
```

```
  _mm_storeu_ps((float*) &((float*)TotalForceDirX_sse)[NeighborIndex], tempX);
  _mm_storeu_ps((float*) &((float*)TotalForceDirY_sse)[NeighborIndex], tempY);
  _mm_storeu_ps((float*) &((float*)TotalForceDirZ_sse)[NeighborIndex], tempZ);

  // Process vertices to Southeast
  NeighborIndex = Index + stride + 1;
  tempX = _mm_loadu_ps((float*) &((float*)VertexX_sse)[NeighborIndex]);
  tempY = _mm_loadu_ps((float*) &((float*)VertexY_sse)[NeighborIndex]);
  tempZ = _mm_loadu_ps((float*) &((float*)VertexZ_sse)[NeighborIndex]);

  diffX = _mm_sub_ps(tempX, VertexX_sse[Index4]);
  diffY = _mm_sub_ps(tempY, VertexY_sse[Index4]);
  diffZ = _mm_sub_ps(tempZ, VertexZ_sse[Index4]);
  diffX = _mm_sub_ps(diffX, RestLengthXSE_sse);
  diffY = _mm_sub_ps(diffY, RestLengthYSE_sse);
  diffZ = _mm_sub_ps(diffZ, RestLengthZSE_sse);

  diff_forceX = _mm_and_ps(WriteMask, _mm_mul_ps(Elasticity_sse, diffX));
  diff_forceY = _mm_and_ps(WriteMask, _mm_mul_ps(Elasticity_sse, diffY));
  diff_forceZ = _mm_and_ps(WriteMask, _mm_mul_ps(Elasticity_sse, diffZ));

  TotalForceDirX_sse[Index4] = _mm_add_ps(TotalForceDirX_sse[Index4], diff_forceX);
  TotalForceDirY_sse[Index4] = _mm_add_ps(TotalForceDirY_sse[Index4], diff_forceY);
  TotalForceDirZ_sse[Index4] = _mm_add_ps(TotalForceDirZ_sse[Index4], diff_forceZ);

  tempX = _mm_loadu_ps((float*) &((float*)TotalForceDirX_sse)[NeighborIndex]);
  tempY = _mm_loadu_ps((float*) &((float*)TotalForceDirY_sse)[NeighborIndex]);
  tempZ = _mm_loadu_ps((float*) &((float*)TotalForceDirZ_sse)[NeighborIndex]);
  tempX = _mm_sub_ps(tempX, diff_forceX);
  tempY = _mm_sub_ps(tempY, diff_forceY);
  tempZ = _mm_sub_ps(tempZ, diff_forceZ);

  _mm_storeu_ps((float*) &((float*)TotalForceDirX_sse)[NeighborIndex], tempX);
  _mm_storeu_ps((float*) &((float*)TotalForceDirY_sse)[NeighborIndex], tempY);
  _mm_storeu_ps((float*) &((float*)TotalForceDirZ_sse)[NeighborIndex], tempZ);
}

k = Deform->NumCols - 1;
for(j=0, Index = i*stride, Index4 = (i*stride)>>2; j<k;  j+=4, Index += 4, Index4++)
{
  FixedMask = _mm_cmpeq_ps(((__m128*)Deform->Fixed)[Index4], AllZeros);
  FixedCompare = _mm_movemask_ps(FixedMask);
  if(FixedCompare != 0)
  {
```

```
TotalForceDirX_sse[Index4] = _mm_add_ps(TotalForceDirX_sse[Index4],
                                        _mm_add_ps(ExternalForceDirX_sse[Index4],
                                                   ConstantForceDirX_sse[Index4]));

TotalForceDirY_sse[Index4] = _mm_add_ps(TotalForceDirY_sse[Index4],
                                        _mm_add_ps(ExternalForceDirY_sse[Index4],
                                                   ConstantForceDirY_sse[Index4]));

TotalForceDirZ_sse[Index4] = _mm_add_ps(TotalForceDirZ_sse[Index4],
                                        _mm_add_ps(ExternalForceDirZ_sse[Index4],
                                                   ConstantForceDirZ_sse[Index4]));

AccelerationX_sse = _mm_div_ps(TotalForceDirX_sse[Index4], Mass_sse);
AccelerationY_sse = _mm_div_ps(TotalForceDirY_sse[Index4], Mass_sse);
AccelerationZ_sse = _mm_div_ps(TotalForceDirZ_sse[Index4], Mass_sse);

VelocityX_sse[Index4] = _mm_add_ps(VelocityX_sse[Index4],
                                   _mm_mul_ps(AccelerationX_sse, DeltaT_sse));
VelocityY_sse[Index4] = _mm_add_ps(VelocityY_sse[Index4],
                                   _mm_mul_ps(AccelerationY_sse, DeltaT_sse));
VelocityZ_sse[Index4] = _mm_add_ps(VelocityZ_sse[Index4],
                                   _mm_mul_ps(AccelerationZ_sse, DeltaT_sse));

VelocityX_sse[Index4] =  _mm_and_ps(VelocityX_sse[Index4], FixedMask);
VelocityY_sse[Index4] =  _mm_and_ps(VelocityY_sse[Index4], FixedMask);
VelocityZ_sse[Index4] = _mm_and_ps(VelocityZ_sse[Index4], FixedMask);

ExternalForceDirX_sse[Index4] = _mm_setzero_ps();
ExternalForceDirY_sse[Index4] = _mm_setzero_ps();
ExternalForceDirZ_sse[Index4] = _mm_setzero_ps();

VertexX_sse[Index4] = _mm_add_ps(VertexX_sse[Index4],
                                  _mm_mul_ps(VelocityX_sse[Index4], DeltaT_sse));
VertexY_sse[Index4] = _mm_add_ps(VertexY_sse[Index4],
                                 _mm_mul_ps(VelocityY_sse[Index4], DeltaT_sse));
VertexZ_sse[Index4] = _mm_add_ps(VertexZ_sse[Index4],
                                 _mm_mul_ps(VelocityZ_sse[Index4], DeltaT_sse));

tempVX = _mm_add_ps(VertexX_sse[Index4], CenterX);
tempVY = _mm_add_ps(VertexY_sse[Index4], CenterY);
tempVZ = _mm_add_ps(VertexZ_sse[Index4], CenterZ);

wv0 = _mm_unpacklo_ps(tempVX, tempVY);
wv1 = _mm_unpacklo_ps(tempVZ, OnePointZero);
```

```
      *(&Deform->WorkingVertex[Index])   =  _mm_shuffle_ps(wv0, wv1, 0x44);
      *(&Deform->WorkingVertex[Index+1]) =  _mm_shuffle_ps(wv0, wv1, 0xEE);

      wv2 = _mm_unpackhi_ps(tempVX, tempVY);
      wv3 = _mm_unpackhi_ps(tempVZ, OnePointZero);

      *(&Deform->WorkingVertex[Index+2]) = _mm_shuffle_ps(wv2, wv3, 0x44);
      *(&Deform->WorkingVertex[Index+3]) = _mm_shuffle_ps(wv2, wv3, 0xEE);
      temp = _mm_sub_ps(OnePointZero, VelocityY_sse[Index4]);
      temp = _mm_max_ps(LowerBound, temp);

      WorkColor_sse[Index]   = _mm_mul_ps(Color_sse[Index],
                                       _mm_shuffle_ps(temp, temp, 0x00));
      WorkColor_sse[Index+1] = _mm_mul_ps(Color_sse[Index+1],
                                       _mm_shuffle_ps(temp, temp, 0x55));
      WorkColor_sse[Index+2] = _mm_mul_ps(Color_sse[Index+2],
                                       _mm_shuffle_ps(temp, temp, 0xAA));
      WorkColor_sse[Index+3] = _mm_mul_ps(Color_sse[Index+3],
                                       _mm_shuffle_ps(temp, temp, 0xFF));
      // set alpha back to original value
      ((float*)&WorkColor_sse[Index])[3]   = ((float*)&Color_sse[Index])[3];
      ((float*)&WorkColor_sse[Index+1])[3] = (float*)&Color_sse[Index+1])[3];
      ((float*)&WorkColor_sse[Index+2])[3] = ((float*)&Color_sse[Index+2])[3];
      ((float*)&WorkColor_sse[Index+3])[3] = ((float*)&Color_sse[Index+3])[3];

      VelocityX_sse[Index4] = _mm_mul_ps(VelocityX_sse[Index4], DampingFactor_sse);
      VelocityY_sse[Index4] = _mm_mul_ps(VelocityY_sse[Index4], DampingFactor_sse);
      VelocityZ_sse[Index4] = _mm_mul_ps(VelocityZ_sse[Index4], DampingFactor_sse);
    }
  }
}
}
```